CS 610 104 Programming Project (Spring 2017)
Sparse Matrix Data Type

A **matrix** $A_{m,n}$is a rectangular array of *m* x *n* numbers (*m, n* > 0). Matrices play an important role in computer science. They are used in algorithms for numerical problems, graphs, pattern recognition, image processing, etc. A **sparse matrix** is a matrix where a large number of the elements have a value of zero. The naïve approach of implementing a sparse matrix is to use a 2D array. However, when working with very large sparse matrices with thousands (or millions, or billions) of rows and columns, significant space is wasted by storing zero values. The matrices below, *A* and *B*, are two small examples of sparse matrices (though they are not very sparse).

$$A = \begin{bmatrix} 3 & 0 & 0 & 5 \\ 0 & 2 & -7 & 0 \\ 5 & -3 & 0 & 0 \\ 0 & 6 & 0 & -5 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

There are several mathematical operations defined for a sparse matrix data type, including **scalarMultiply(*c*), add(*M*), subtract(*M*), multiply(*M*), power(*p*),** and **transpose().**

There are many different data structures that can be used to implement a sparse matrix. In this project you will use an array-based data structure. The array representing the sparse matrix will store *Matrix Entries*, each of which consists of a value (a non-zero integer), a row (a positive integer), and a column (a positive integer). The *Matrix Entries* are stored in the array in non-decreasing order according to their row and are subsorted in non-decreasing order according to their column. The table below illustrates matrix A in this data structure. Note that the size of the array is intended to be **significantly** smaller than *m* * *n* in most cases.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
row	1	1	2	2	3	3	4	4
column	1	4	2	3	1	2	2	4
value	3	5	2	-7	5	-3	6	-5

The end goal of this project is to have a program that can represent any *m* x *n* matrix and be able to perform mathematical operations on sparse matrices. There are several operations defined for a sparse matrix data type.

SparseMatrix(S)	Initializes a sparse matrix using a comma-delimited string S in the following format: "*n*r*i*c*j*, *n*r*i*c*j*, *n*r*i*c*j*, ...", where *n* is an integer indicating the value, *i* is an integer indicating the row, and *j* is an integer indicating the column. The entries in S **are in no pre-defined** order, so you will need to take that into account. Matrix A from Page 1 could be stored as: -5r4c4, 5r1c4, 2r2c2, 5r3c1, -3r3c2, 6r4c2, -7r2c3, 3r1c1 Note that the size of the matrix *m* x *n* can be determined based on the entries in S. In the above example you can find m = n = 4.
SparseMatrix(*m, n*)	Initializes an empty sparse matrix with *m* rows and *n* columns. To be used when initializing a result Sparse Matrix in add, subtract, multiply, etc.
print()	Prints the Sparse Matrix to standard output. Note that zeros need to be printed in print(). Additionally, your print operation will need to print the matrix so it looks like a matrix. If there are more than six rows only print the first two rows and the last two rows, separated by … in one intermediate row. If there are more than six columns only print the first two columns and the last two columns, separated by a … in one intermediate column. **NOTE:** print only prints, it ***does not*** return any value (e.g., a string).

The mathematical operations for the sparse matrix are defined in the following table. Note that all of the mathematical operations are **immutable**. They do not modify the given sparse matrix **A**. Instead they create a new sparse matrix **R** to store the result of the operation.

equals(M)	Returns a boolean indicating if **A** = **M**
scalarMultiply(c)	Returns the result of multiplying the matrix by an integer constant *c*: *c***A**
add(M)	Returns the result of adding the sparse matrix M to the sparse matrix, if M is the same size as the sparse matrix: **A + M**
subtract(M)	Returns the result of subtracting the sparse matrix M from the sparse matrix, if M is the same size as the sparse matrix: A - M
multiply(M)	Returns the result of multiplying the sparse matrix by M: **A * M**
power(p)	Returns **A**^p, where p is an integer >= 1. No more than 2lg(p) calls to multiply are permitted.
transpose()	Returns the transpose of **A**

Note that, unlike initialization from a string, when a SparseMatrix is created as a result of a mathematical operation (e.g., add, multiply) you do not necessarily know how large the array will be. In such a case you will start the array at some initial size and increase its size in an efficient manner (e.g., double its size after a certain number of Matrix Entries have been added to the array).

This project is separated into two major steps. Both steps are broken into several parts. You cannot move on to the second step of the project until every part of the first step is implemented correctly or you are cleared to move onto the next step (you will be explicitly told that you ***cannot*** move on when you receive a grade for the first step). Correcting a part of a step after a grade was given will result in no additional credit. **You must execute all of the required test cases for a given step or you will receive a grade of zero for that step.**

Step 1 (45 points)	**Array Implementation and Basic Mathematical Operations**	**Due Date: 3/31**
Implement SparseMatrix(S) (12 pts) and SparseMatrix(m, n) (2 pts) Implement print() (12 pts) Implement equals(M) (8 pts) Implement scalarMultiply(c) (10 pts) Run the required Test Cases for Step 1 (6 pts)		
Step 2 (55 points)	**Mathematical Operations**	**Due Date: 4/28**
Implement add(M) and subtract(M) (15 pts total) Implement multiply(M) (20 pts) Implement power(p) (10 pts) Implement transpose (10 pts) Run the required Test Cases for Step 1 and Step 2. Calculate the running time of each individual test case. (10 pts)		

Your submission will be graded according to the correctness and efficiency of your implements. Operations such as print, equal, scalar multiply, add, and subtract should take O(nm) time. The running times of multiply, power, and transpose should be near the asymptotic lower bound.

Due Dates and Late Submissions

Each step is due by 11:59:59PM on the given due date. You may submit a step up to one week late at a penalty of two points per day. For example, if you submit Step 1 three days late the maximum possible grade for Step 1 will be 39/45. If you cannot complete a step within one week of the due date then you will need to meet with me privately during my office hours to discuss your progress.

Implementation Guidelines

You will receive **no credit** for your submission if you do not follow these guidelines. **READ THESE CAREFULLY.** If you are unsure if you are following these guidelines do not hesitate to contact me or the TA.

1. You can use either C++ or Java. You **must** use the skeleton project code provided on Moodle.
2. Your Sparse Matrix ADT **must** be implemented using the described array-based data structure.
3. You **must** implement all of your own data types and data structures. You cannot use any built in data types (aside from strings for initialization and arrays) or include any data type libraries.
4. The correct implementation of the required operations is dependent on the correct implementation of the specified data types and data structures. If you have the required functions working with incorrect data types of data structures you will receive no credit.
5. A Sparse Matrix can **never** be converted into a string other than for print(). **No exceptions.** Additionally, strings can only be used in main and SparseMatrix(S) and print().
6. A Sparse Matrix can never be converted into another data structure or data type.
7. Use of 2D arrays is **strictly** prohibited. **No exceptions.**
8. Your Sparse Matrix can never store a zero value.
9. All of the specified Sparse Matrix mathematical operations are immutable operations. The Sparse Matrix, and the operation argument(s), must be unchanged by these operations.
10. Every step **must** automatically run the required test cases. There is no user input for this program.

You can implement additional methods as needed, as long as they do not violate any of the above guidelines. You can also name any function or variable as you wish, as long as it's a meaningful name.

Submission Guidelines

- Submission will be done through Moodle.
- Submit only what is required for the current step.
- Submit **only** the source code for your submission. Only submit .cpp/.cc/.h/.hh/.c/.java files. Combine multiple such files into a single Zip file.
- Your code **must** compile and execute on NJIT's AFS computers (afsconnect1.njit.edu or afsconnect2.njit.edu) using either g++ or javac.
- Test that your program compiles on AFS early and often.
- Do not use non-standard libraries (e.g., conio.h).
- If your Step 1 submission does not compile on AFS you will be given 24 hours to correct the problem with no penalty. After 24 hours it will be considered late. In all other situations, if your submission does not compile on AFS it will receive a grade of zero.
- Mention any specific instructions for compiling in your submission email.
- You can modify a previous step's code for a future step, if necessary.

Academic Honesty Policy

Copying code from the internet, another student, or from any other source (including books, YouTube videos, 8-track tapes, post-it notes, etc.), even if it is just a single line of code, is **strictly** prohibited. Any cases of copying will result in everyone involved being reported to the Dean of Students and everyone involved will receive a zero for the entire project. The course's academic honesty policy, specified in the syllabus, will also be strictly enforced.

- Your code is yours and yours alone. You are responsible for it.
- Work on your own at all times.
- Do not share your implementations or code with other students.
- Do not look at each other's code.
- Do not share your code as a "guide" for other students.
- Do not put your code in a publicly accessible place (e.g., Github, Stack Overflow, Reddit, Pastebin, etc.).
- Do keep a record of your work. If there is any question regarding the legitimacy of your submission have this information available.

Test Cases

To receive any credit your project must automatically run, without user input, the specified test cases for each step. If your submission does not automatically run the required test cases you will receive a grade of zero for the step. Without these test cases the TA will not be able to determine if your project works.

For the test cases you will use several sparse matrices, named A – G, to test your program.

Matrices A and B are the matrices on Page 1 and should be initialized with SparseMatrix(S). Matrices C – G are defined according to the following mathematical functions and need to be automatically initialized by your program. You need to design a way to initialize these matrices without using strings (a separate constructor for these test cases may be a good idea!)

Matrix C is a 5 x 6 matrix defined such that $A_{i,j} = \begin{cases} i * j & (i+j)\%2 = 0 \\ 0 & otherwise \end{cases}$

Matrix D is a 6 x 5 matrix defined such that $A_{i,j} = \begin{cases} i + j & (i*j)\%4 = 0 \\ 0 & otherwise \end{cases}$

Matrix E is a 200 x 200 matrix defined such that $A_{i,j} = \begin{cases} i + 2j & i\%10 = 0 \\ 0 & otherwise \end{cases}$

Matrix F is a 200 x 1 matrix defined such that $A_{i,j} = \begin{cases} 5 * i & i\%5 = 0 \\ 0 & otherwise \end{cases}$

Matrix G is a 30,000 x 30,000 matrix defined such that $A_{i,j} = \begin{cases} i + j & i\,\%\,j = 0 \\ 0 & otherwise \end{cases}$

Step 1 Test Cases

- Initialize matrices A – G.
- Print matrices A – G using print().
- Scalar multiply each matrix by 5 and print the result.
- Apply equals() pairwise among matrices A – G (i.e., compare A to A, A to B, A to C, etc.).

Step 2 Test Cases

- Add each matrix to itself and print the result.
- Subtract each matrix from itself and print the result.
- Scalar multiply each matrix by five and then subtract the matrix (e.g., 5A – A) and print the result.
- Multiply Matrix B with the transpose of Matrix C, Matrix C with Matrix D, Matrix D with Matrix C, and Matrix E with Matrix F. Print the result for each.
- Multiply Matrix A with itself and Matrix E with itself.
- For Matrices A and E raise each matrix to the power of 5 and 25 using power().
- Compute the transpose of each matrix and print the result.
- Multiply each matrix with its transpose and its transpose with itself. Print the result for each.
- Compute the running time for each individual test case in Step 2 (e.g., each time you call add, subtract, or multiply for a test case, determine its running time in nanoseconds (Java) or ticks (C++).